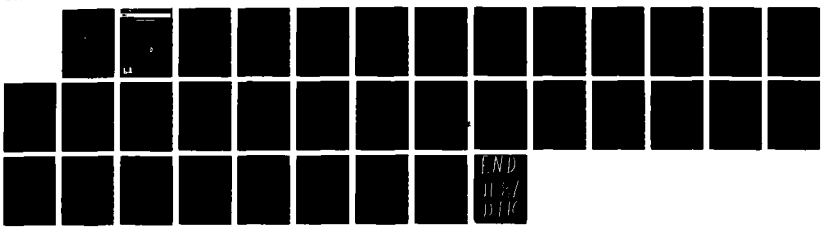
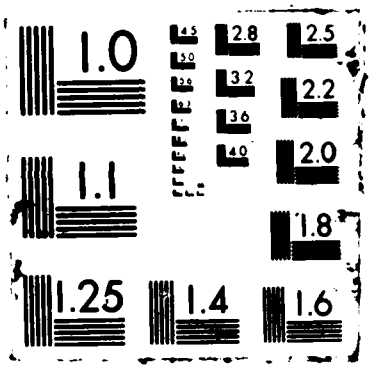
AD-A184 890 A METHODOLOGY FOR PARTITIONING REAL-TIME ADA (TRADENAME) SOFTWARE FOR DIS..(U) GTE LABS INC WALTHAM MA COMPUTER AND INTELLIGENT SYSTEMS LAB.. 1/1

UNCLASSIFIED A CHOW ET AL. SEP 86 N00014-85-C-0796 F/G 12/5 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
32
36
40

DTIC FILE COPY

(10)

GTE Laboratories Incorporated

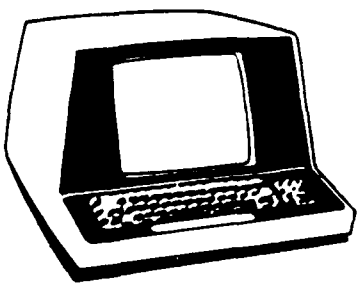
AD-A184 890

FINAL REPORT

A METHODOLOGY FOR PARTITIONING REAL-TIME ADA®[1] SOFTWARE FOR DISTRIBUTED TARGETS[2]

DTIC ELECTE SEP 1 8 1987 S D C&D

DISTRIBUTION STATEMENT A
Approved for public release
Distribution Unlimited



Computer and Intelligent Systems Laboratory
GTE Laboratories Incorporated
40 Sylvan Road
Waltham, MA 02254

87 9 14 024

1133

(10)

FINAL REPORT
Contract No. N00014-85-C-0796

A METHODOLOGY FOR PARTITIONING REAL-TIME ADA®[1] SOFTWARE FOR DISTRIBUTED TARGETS[2]

by

A. Chow
M. Feridun

DTIC
ELECTE
SEP 1 8 1987
S D
CSD

September 1986

Computer Science Laboratory
GTE LABORATORIES INCORPORATED
40 Sylvan Road
Waltham, Massachusetts 02254

---

[1]Ada is a registered trademark of the U.S. Department of Defense, A.J.P.O.

[2]This research has been partially supported by the Office of Naval Research under grant number N00014- 85-C-0796.

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

TABLE OF CONTENTS

1.0 Introduction 1

2.0 Partitioning Problem and Algorithms 4

3.0 Partitionable Units 8

4.0 Partition Model 10
4.1 Interunit Communication 10
4.2 Computational Complexity of the Units 15

5.0 An Example 17

6.0 Conclusion 22

QUALITY INSPECTED 2

Accesion For
NTIS CRA&I ☑
DTIC TAB ☐
Unannounced ☐
Justification
By per ltr.
Distribution /
Availability Codes
Dist | Avail and / or Special
A-1

1.0 INTRODUCTION

Task allocation, the assignment of tasks to processors, is an important problem in the design of distributed real-time systems. A task allocation scheme is required in order to produce a feasible partition of tasks across processors in the system, and to ensure high performance, especially for systems with real-time operational requirements. Several researchers have studied the task allocation problem for distributed systems; [Chu80] contains a survey of various approaches.

One of the problems of current interest in real-time system design is the development of real-time Ada software for distributed systems. Several approaches have been proposed and are being studied; a survey can be found in [Arm84]. The approaches can be characterized as either

- ***source code allocation***, the development of multitasking Ada software, which is then partitioned ; this approach allows development and testing of the software as a whole before allocation;

- ***target-code allocation***, where a compiler is responsible for performing task allocation, perhaps with some user-imposed contraints; or

- ***separate program development***, where allocation decisions are made early in the development phase, and separate programs are developed.

The traditional approach of developing separate source programs for each processor in the distributed system requires the system designer to make early decisions on allocation, taking into account resource and performance constraints. This approach, however, increases the difficulty of software reallocation in later phases of the

software life cycle. Target code allocation schemes require a distributed target compiler that is used to generate separate object code files in allocating target code for each processor. There are two ways that the compiler can partition Ada application software: (1) being informed via pragmas about a predetermined partition scheme; or (2) analyzing the application software, and then applying a partitioning algorithm. The compiler required for this allocation scheme is complex, difficult to design, and presently not available.

The source code allocation approach has a number of important advantages. In an allocation scheme, it is preferable to place minimum design restrictions on software development, especially since the target architecture may, in most cases, not be known. It is also preferable to minimize the burden on the compiler. Additionally, since the underlying system constraints may vary, a good partition can be achieved only through iteration, and therefore, creating new partitions should be inexpensive.

A source code allocation approach that meets the above objectives was adopted by GTE Strategic Systems Division in its multicomputer software technology for Ada. This approach allows an application to be developed and tested as a single multitasking Ada program on the APSE (Ada Programming Support Environment), and then partitions and distributes the tested software to the distributed targets. Program partitioning is done at the source level, and the distributed software modules are compiled on the target machines.

GTE Laboratories researched the development of a methodology for partitioning Ada source code to execute in a distributed environment. Two major tasks were involved in the development of such a methodology:

1. the formulation and selection of parameters that can be derived from the Ada source code, to be used in the partitioning process; and

2. the development of an efficient partitioning algorithm.

The approach taken closely follows that described in a previous report [GTE85]. The basic goal of the approach is to transform a given Ada program into a graph based representation, and then to apply a partitioning algorithm for task allocation. The graph based representation is similar to the specification schemes being investigated at the University of Texas at Austin [Mok84, Mok85].

In this paper, we describe the research efforts towards achieving the above tasks. In section 2, we describe the partition problem and describe an algorithm for it. In section 3, we define partitionable units in Ada software. In section 4, we enumerate parameters derivable from Ada source code to be used in partitioning. In section 5, we present an Ada example. Section 6 concludes the report with a suggestion for a future research direction.

2.0 PARTITIONING PROBLEM AND ALGORITHMS

The problem of allocating Ada source over distributed targets can be formulated as a graph partitioning problem. For purposes of task allocation, an Ada program can be represented by a graph G = (V,E) as follows:

- the vertices of the graph G, i.e., the set V, represents the partitionable Ada program units; and
- the communication or dependency between units is represented by the edge set E.

Given this representation of the Ada program, weights are assigned to the vertices and edges; a weight w(v) for vertex $v \in V$ represents execution characteristics of the Ada partitionable unit, obtained from such parameters as computation , memory and similar resource requirements. The weight $w(e_{ij})$ assigned to the edge between unit (vertex) v_i and unit (vertex) v_j represents the total communication cost between the two units; this weight is obtained from such parameters as the number of data elements transferred in a communication, and the number of messages required per transaction.

The partitioning problem for graph G = (V,E) can be formulated as follows: determine a partition of V into m disjoint subsets $V_1, V_2, \ldots, V_m$ such that

(1) $J \leq \sum_{v \in V_i} w(v) \leq K$, $1 \leq i \leq m$, for some constants J and K

and

(2) $I(V_i,V_j) = \sum_{e \in E_{ij}} w(e) \leq L$, for some constant L, $E_{ij} \subseteq E$ and

$(v',v'') \in E_{ij} \Rightarrow v' \in V_i, v'' \in V_j$ and $V_i \neq V_j$.

The partitioning problem, as formulated, aims at reducing the communication cost $I(V_i, V_j)$ between the partitioned clusters, and places a load balancing constraint on the clusters. These objectives are appropriate for the Ada partitioning problem as system performance is affected by factors such as interprocessor communication delays, processor load, and the amount of parallelism that can be exploited.

The partitioning problem as formulated has been shown to be NP-complete [Gar79]; however, there are partitioning algorithms that use heuristics to obtain close to optimal results with acceptable algorithm performance [Pri84,Ker69].

The heuristic techniques reported in the literature can be classified into three categories [Lin81]: (1) constructive initial assignment, (2) iterative assignment-improvement, and (3) branch and bound technique.

The ***constructive initial assignment*** techniques are based on the concept of assigning one unit at a time to a particular processor until all the units are assigned. Algorithms vary on the order in which the units are assigned and the criteria used to select the processor.

The ***iterative assignment-improvement*** techniques start with an initial assignment of units to processors and generate the next assignment by making a small improvement to the initial assignment [Ker69]. The algorithms terminate when no improvement can be discovered or after a predetermined number of iterations.

The ***branch and bound*** methods are based on the concept of doing an implicit search of a decision tree. Algorithms use different heuristic methods for deciding which branch in the decision tree to follow and for pruning possible solutions.

The iterative assignment-improvement algorithms are used more than the other two techniques. In general, the branch and bound algorithms are too slow for large applications and the constructive initial assignment algorithms do not generate partitions that are as good as the other two techniques.

It is expected that the number of vertices in a graph obtained from Ada source will be large, and therefore heuristics will need to be developed not only for creating good partitions, but also for partitioning in an acceptable amount of time.

In the remainder of this section, we describe an algorithm based on a well-known graph partitioning algorithm by Kernighan and Lin [Ker69]. The algorithm uses the iterative assignment-improvement techniques.

The main idea of the algorithm is to start with an initial partitioning into m subsets and by repeated application of iterative-assignment-improvement techniques to pairs of subsets, to achieve a near-pairwise-optimal state.

To obtain the initial partition, the units are assigned, one by one, to the subset with least weight first, provided the balancing inequalities are satisfied. Next, the interactive assignment-improvement algorithm is applied to every pair of subsets and it works as follows:

Let A, B be two arbitrary subsets, the algorithm identifies X and Y, subsets of A and B respectively, such that interchanging X and Y produces A^* (= A -X + Y) and B^* (= B -Y + X) that satisfy the following conditions:

(1) $J \leq \sum_{v \in A^*} w(v) \leq K$,

$J \leq \sum_{v \in B^*} w(v) \leq K$, and

(2) $I(A^*, B^*) < I(A, B)$.

The algorithm finds X and Y by sequentially identifying their elements without considering all possible choices. A pair of units (a, b), where $a \in A$ and $b \in B$, are selected to yield the largest possible reduction in the interprocessor communication cost from a single interchange. We refer to this reduction in the communication cost as the gain from interchanging a and b, and we denote this gain g(a,b). Mathematically, the gain g(a,b) is

$$\left(\sum_{y \in B'} w((a, y)) - \sum_{x \in A'} w((a, x))\right) + \left(\sum_{x \in A'} w((b, x)) - \sum_{y \in B'} w((b, y))\right) \text{ where}$$

$A' = A - \{a\}$ and $B' = B - \{b\}$.

The gain g(x,y) is defined to be zero if interchanging x and y upsets the balancing inequalities. The elements a and b are interchanged to form A_1 ($= A - \{a\} + \{b\}$) and B_1 ($= B - \{b\} + \{a\}$). The elements a and b are then eliminated from further consideration for exchange. The exchange procedure continues until all units have been exhausted, or there is no further possible exchange that will yield a positive gain.

3.0 PARTITIONABLE UNITS

In our framework of source code partitioning, we discuss what constitutes a partitionable unit of an Ada program. Since the partitioned software has to be compiled on each processor, the partitioned units must be separately compilable. In Ada, there are four kinds of program units that can be separately compiled. They are tasks, subprograms, packages, and generic units.

Subprograms are the basic executable units of Ada programs. They can be procedures or functions. A subprogram communicates with outside entities via global declarations or parameter passing upon its invocation and termination.

In Ada, a collection of logically related entities can be encapsulated in a package. A package allows its entities to communicate with an entity outside the package via global declaration or by the import and export mechanism. The entities declared in the visible part of the package specification may be used outside the package. And entities in another package may be used by establishing the visibility through the with clause.

Unlike subprograms and packages, tasks operate in parallel with other program units. The main program unit is implicitly considered to be a task. In Ada, task interaction is handled by treating each task as a communicating sequential process [Hoa78]. The tasks are synchronized in time when they communicate. The explicit synchronization is known as a rendezvous. Similar to package specification, a task specification defines the communication entries available to other tasks.

These three kinds of program units can be introduced in the declarative part of any unit. This makes the communication among units non-trivial. We discuss this in a later section.

Some distributed Ada systems allow partitioning on task boundaries only. That is, this approach appears to have achieved a synergy between Ada's units of concurrency, the task, and the underlying system's unit of concurrency, the processor. However, this approach requires all code being partitioned to be encapsulated by a task. This requires early partitioning to make sure that tasks are designed at the appropriate place. This does not meet our objective of making minimum design restrictions. In some applications limiting interprocessor interface to only task rendezvous may be unnatural. The interprocessor interface may be better represented as a call to a procedure inside a package and not a call to an entry for a task.

We propose that partitioning be allowed on these three kinds of program unit boundaries. We do not explicitly include generic units as partitionable units. We can view generic packages/subprograms and their instantiations as packages/subprograms in the partitioning scheme. Since the partitioned units have to be compiled on the target machines, we may require the partitionable units to be designed as Ada compilation units for distribution purposes. This does not impose any syntactic restriction or any design restrictions since Ada program units can be submitted as separate compilation units or as one compilation. However, it must be understood that a compilation unit does not have to be a partitionable unit. In Ada, each compilation unit specifies the separate compilation of a construct that can be a subprogram declaration or body, a package declaration or body, a generic declaration or body, or a generic instantiation. A compilation unit can also be the body of a task unit.

4.0 PARTITION MODEL

We have discussed the general partition problem and our proposed partitionable units in Ada in previous sections. We now propose a model for partitioning an Ada program for distributed targets.

The first step in our modeling is to represent the interunit communication as a graph G = (V,A), where V is the set of vertices representing the partitionable units of an application, and A is the set of arcs representing the communication. Our next steps will be examining how to assign weights to the vertices and the arcs.

4.1 INTERUNIT COMMUNICATION

A unit can be a subprogram, a task, or a package of data objects. There are four kinds of communication among these units.

The first is ***subprogram invocation.*** A subprogram's execution is invoked by a subprogram call from another subprogram or a task. After the association between formal parameters and actual parameters is established, execution control is passed to the called subprogram. Upon completion, control is returned to the caller. The subprogram invocation follows a single thread of control. The communication cost is incurred at invocation and completion.

The second kind of interunit communication is ***task rendezvous.*** Different tasks execute independently, except when they communicate. A task entry can be called by another task. Communication is established when the called task accepts the call. If the entry has parameters, values are communicated between the tasks. After this synchronization, the task issuing the entry call and the task accepting the call continue their execution independently.

Task activation/termination is another kind of communication related to tasks. This is an implicit communication in the control flow of the task dynamics. The initial part of task execution is called activation. A task is activated as a result of the elaboration (execution) of the declarative part of its parent task or as a result of the allocation of a new task. A task is said to be terminated when it is completed (it finishes its last executable statement) and all its dependents are terminated. Therefore, upon termination, a dependent task needs to communicate its state to its parent. This kind of activation/termination communication occurs between a task and any other kind of partitionable unit.

Data reference/modification is a different kind of interunit communication that is not explicit. A partitionable unit can reference any visible data defined in other units. A data definition in a unit is made visible to another unit either by scope rules or with clauses. Data reference/modification is purely data flow; there is no control flow involved.

Two units with any of these four kinds of communication will experience some network delay when they are allocated to different processors. Over the same network, different kinds of communication take different amounts of time. We discuss the weights on these kinds of communication in more detail below.

Although communication is bidirectional, we like to assign direction to it for analysis purposes. We say a communication is from unit A to unit B if unit A initiates the communication.

We assign a weight to every communication from unit A to unit B if they are assigned to different processors. The weight of an interprocessor communication depends on the number of messages required for such a communication. In the case of

subprogram invocation and task rendezvous, the number of arguments in the call has to be taken into account for the weight of the communication.

For a call to a subprogram on a different processor, two messages are needed: a "call" message containing the IN parameters, if there are any, and a "return" message containing any OUT parameters:

subprogram call

s_call	requesting unit	subprogram id	input parameters

subprogram return

s_return	requesting unit	output parameters

We assign a weight of

3 + number of input parameters

to a subprogram "call" message and a weight of

2 + number of output parameters

to a subprogram "return" message. Thus a ***subprogram invocation*** type of communication is assigned a weight of

5 + number of input parameters + number of output parameters.

In addition to the "call" and "return" messages, there are two more messages needed for each normal task rendezvous [Wea84]. After a "call" message is sent to the accepting task and when the accepting task is ready to accept the task, it returns an "accept" message to the calling task. If the calling task still desires rendezvous, it returns a "confirm" message to the accepting task. The message passing required for task rendezvous is dipicted in figure 1. The "accept" and "confirm" messages are less complex than the "call" and "return" messages that contain IN and OUT parameters:

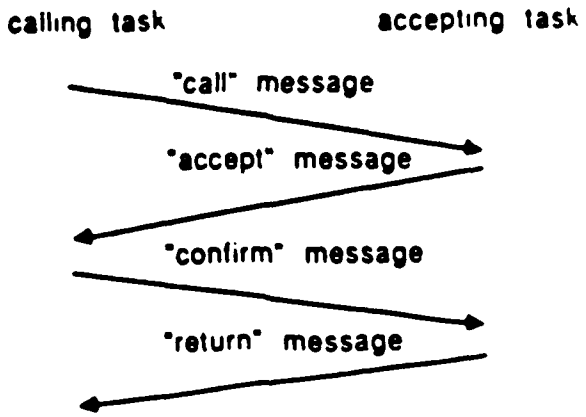


Figure 1. Messages Required for Task Rendezvous

entry call

e_call	requesting unit	entry id	input parameters

accept

accept	requesting unit

confirm

confirm	requesting unit

return

return	requesting unit	output parameters

Weights are assigned to each of these four messages in the same manner as for subprogram invocation messages. The weight assigned to a ***task rendezvous*** type of communication is

9 + number of input parameters + number of output parameters.

In the event of elaboration, a task sends an "elaborate" message to all its dependent tasks and waits for an "active" message from each of them. When a task completes its last executable statement, it waits for a "terminate" message from each of its dependents before it terminates. Therefore, there are totally three messages between a parent and each of its dependents required for activation and termination:

task activation

activation	requesting unit	task id

task activation return

a_return	requesting unit

task termination

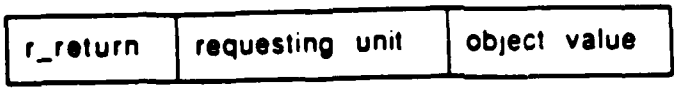
termination	parent unit

A ***task activation/termination*** type of communication is thus assigned a constant weight of 7.

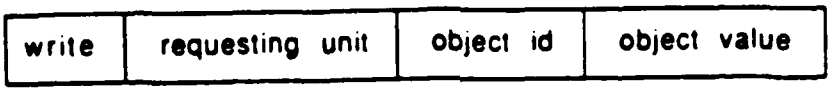
For data reference/modification between two units on different processors, a "request" message is sent from the initiator and a "response" message is returned:

read

read return



write



write return

A ***data reference/modification*** type of communication is assigned a weight of

5 + size of the data object.

Knowing the weight of each kind of communication, the weight of an edge from unit A to unit B is computed as the sum of the weights of all communication from A to B.

4.2 COMPUTATIONAL COMPLEXITY OF THE UNITS

Several program complexity metrics have been developed for various purposes such as maintainability and understandability. For partitioning purposes, we are interested in the computational complexity of a unit. The complexity measure includes the unit's time and space requirements. Knowing the complexity of each unit, we may be able to achieve better load balancing for the processors in the system. We use a simple definition of load balancing. Load balancing is an assignment of units to processors such that the time and space requirements are evenly distributed to each processor in the system.

A unit, except a package of data, contains a code portion and a data portion. A suitable metric for measuring the space requirements of the code portion of a unit might be the number of machine instructions. However, the number of machine instructions generated from Ada source is compiler dependent. In general, the expansion ratio of the number of machine instructions per line achieved by a compiler is not known. GTE-GS SSD has done some work in measuring empirically the expansion ratio of a group of compilers [Che86]. Since we are using a less strict definition of load balancing; that is, we are not aiming at an optimal assignment, the number of source lines could be a good estimate of the space requirement for a unit's code portion. Similar to code space requirements, a unit's data storage requirement is compiler dependent. At this point, we are going to use a set of assumptions about Ada data type storage requirements.

The time complexity of a task or subprogram unit that does not contain a loop statement can be measured by the number of machine instructions generated for the unit. To analyze a unit with loop statements is nontrivial. A loop statement without an iteration scheme can be repeatedly executed until a transfer of control occurs. In most cases, the number of times a loop is going to be executed cannot be predicted until the transfer of control occurs. Even for a loop statement with a "while" iteration scheme, it is difficult to estimate the number of iterations. We can only be certain of the number of iterations in a loop statement with a "for" iteration scheme. We can view all loop statements with or without iteration schemes as a sequence of code to be executed periodically. Therefore, the time requirement of a unit can be estimated by the number of source lines without regard to loop statements.

5.0 AN EXAMPLE

In this section we examine an Ada embedded system for monitoring temperatures as described in [Boo83]. In the appendix, we include the program that was taken in its entirety from chapter 18 of [Boo83]. We formulate the partitioning problem and apply the partitioning algorithm described in section 3 to it.

The Ada program consists of the main program, four major tasks and a number of I/O packages. We center our partitioning problem on the main program and the four tasks, and ignore the I/O packages to simplify the discussion. The partitioning problem we are addressing is as follows:

Find a partition of the five Ada units: TIMER, ALARM, COLLECTION_OF_SENSORS, RECORDING_DEVICE, and MONITOR_TEMPERATURES into two subsets V_1 and V_2 such that

(1) $36 \leq \sum_{v \in V_i} w(v) \leq 190$

where $\left(\sum_{v \in V} w(v)\right)/2 - \max_{v \in V}(w(v)) = 36$

and $\left(\sum_{v \in V} w(v)\right)/2 + \max_{v \in V}(w(v1)) = 190$

(2) $I(V_1, V_2)$, the interprocessor communication cost is near-minimal.

The complexities of the units are

Unit	Source line of code
TIMER	48
ALARM	30
COLLECTION_OF_SENSORS	53
RECORDING_DEVICE	19
MONITOR_TEMPERATURES	77

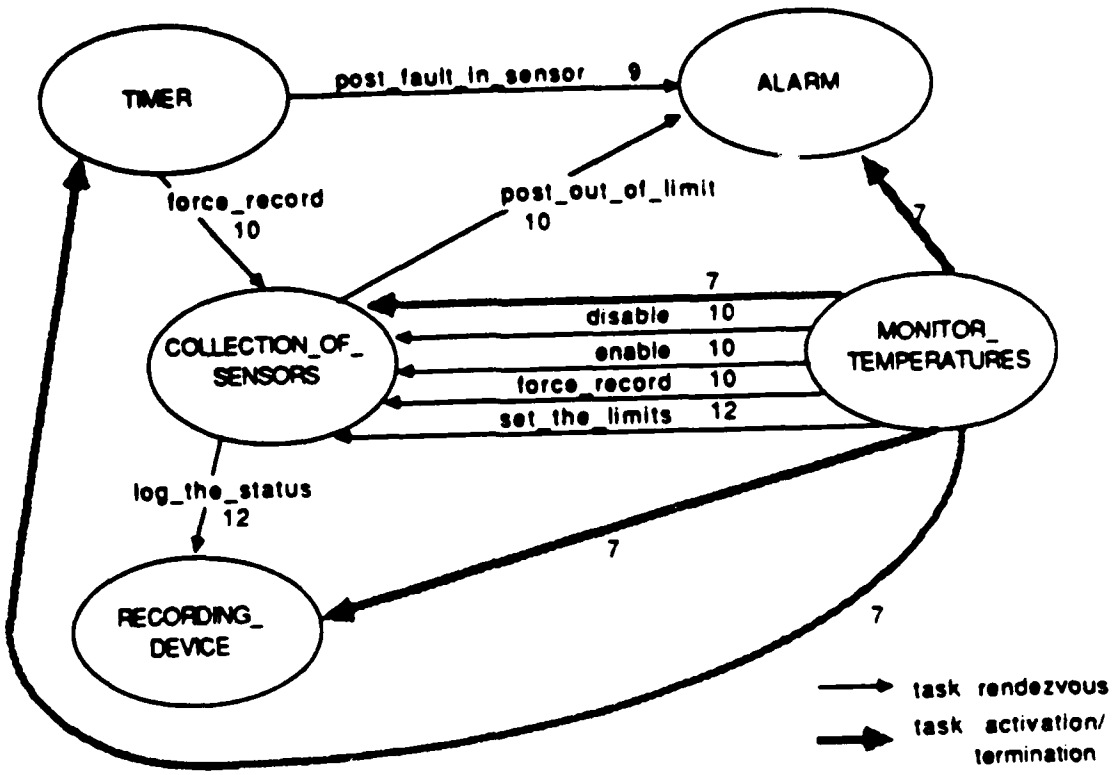


Figure 2. Interunit Communication

We sum up the interunit communications and produce the graph representation of the program as shown in figure 3.

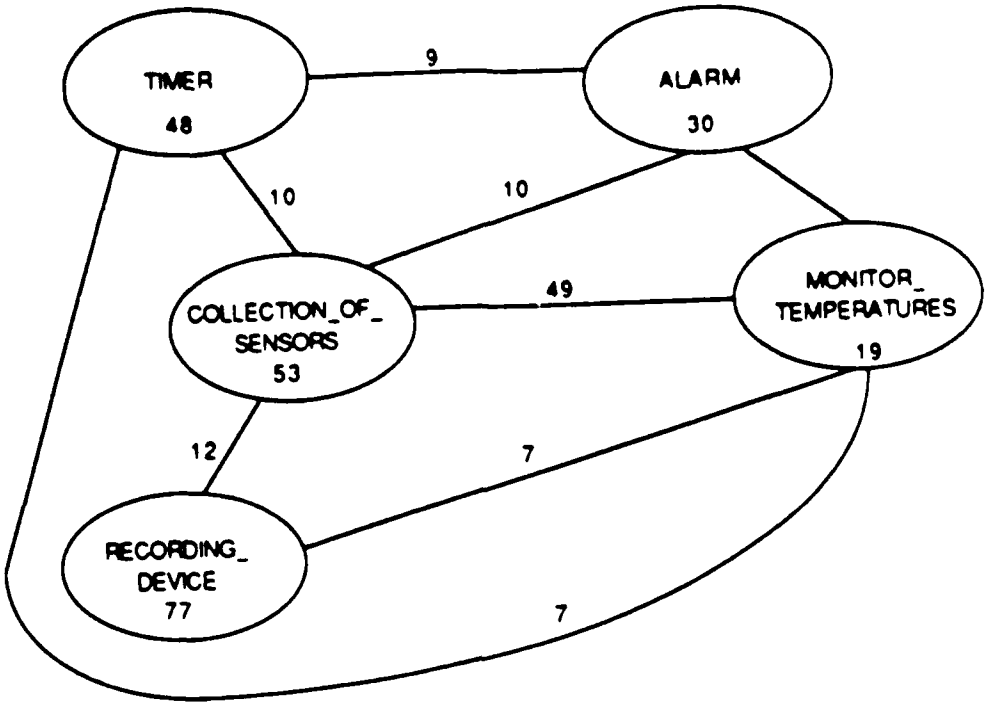


Figure 3. Graph representation of the program for partitioning

In total, there are sixteen ways to partition five units. Three of these partitions are not qualified as they fail to satisfy the balancing constraints. The thirteen possible partitions and their corresponding communication costs are

V_1	V_2	$I(V_1,V_2)$
T	A,C,R,M	26
T,A	C,R,M	34
T,R	A,C,M	45
T,C,M	A,R	52
T,A,R	C,M	53
T,A,C,R	M	70
T,A,R,M	C	81
T,M	A,C,R	82
T,A,C	R,M	82
T,C,R	A,M	82
T,C	A,R,M	87
T,R,M	A,C	87
T,A,M	C,R	88

Where,

T = TIMER

A = ALARM

C = COLLECTION_OF_SENSORS

R = RECORDING_DEVICE

M = MONITOR_TEMPERATURES

We obtain an initial partition by assigning units to V_1 or V_2, depending which has less weight. So V_1 = {TIMER, RECORDING_DEVICE, MONITORING_TEMPERATURE} and V_2 = {ALARM, COLLECTION_OF_SENSORS}. This initial partition has a cost of $I(V_1, V_2)$ = 87. We are searching for the first interchange that will yield a positive reduction in cost. There are six possible pairwise interchanges, and their corresponding gains are

TIMER	< ----- >	COLLECTION_OF_SENSORS	53
RECORDING_DEVICE	< ----- >	COLLECTION_OF_SENSORS	42
MONITOR_TEMPERATURES	< ----- >	ALARM	34
RECORDING_DEVICE	< ----- >	ALARM	11
MONITOR_TEMPERATURES	< ----- >	COLLECTION_OF_SENSORS	5
TIMER	< ----- >	ALARM	0

We pick the pair TIMER and COLLECTION_OF_SENSORS for interchange to give a maximum positive reduction in cost and form V_1' and V_2', where V_1' = {MONITORING_TEMPERATURE, ALARM, RECORDING_DEVICE}, and V_2' = {COLLECTION_OF_SENSORS, TIMER}. Next, we exclude ALARM and TIMER from consideration for exchange. There are then two pairs of units to be considered: (MONITOR_TEMPERATURES, ALARM) and (RECORDING_DEVICE, ALARM). Both pairs yield negative gains. We stop the interchange process. The final partition is thus {V_1', V_2'} which has an interprocessor communication cost of 34; that is the second best partition.

6.0 CONCLUSION

In this paper, we have discussed the problem of partitioning real-time Ada software for distributed targets and adopted source code allocation as an approach to the problem. In this approach, code for an application is developed and tested as a single Ada program, and then partitioned and distributed to distributed targets, where compilation takes place at each location. A partitioning methodology for Ada programs has been outlined.

The example we presented favorably shows the effectiveness of the partitioning methodology. However, its performance has not been assessed formally.

We have used a simple definition of load balancing. This definition is acceptable only for 0(n) type programs. It is necessary to provide a more accurate measure of load balancing for other types of programs.

In this report, we have considered the complexities of the partitionable units and message complexities of interunit communications. We did not consider parameters such as channel capacities nor complexities of the processors. We feel that there is a need to research in identifying important parameters for partitioning.

APPENDIX A

```
with TEXT_IO, SYSTEM;
use  TEXT_IO;
procedure MONITOR_TEMPERATURES is
  --
  type COMMAND       is (DISABLE,            ENABLE,
                         RECORD_STATUS,      SET_LIMITS);
  --
  type SENSOR_NAME   is (LOBBY,              MAIN_OFFICE,     WAREHOUSE,
                         STOCK_ROOM,         TERMINAL_ROOM,   LIBRARY,
                         COMPUTER_ROOM,      LOUNGE,          LOADING_DOCK,
                         CLEAN_ROOM);
  type SENSOR_STATE  is (DISABLED, ENABLED);
  type SENSOR_VALUE  is delta 0.5 range 0.0   100.0;
  --
  package COMMAND_IO          is new ENUMERATION_IO(COMMAND);
  use     COMMAND_IO;
  package SENSOR_NAME_IO      is new ENUMERATION_IO(SENSOR_NAME);
  use     SENSOR_NAME_IO;
  package SENSOR_VALUE_IO     is new FIXED_IO(SENSOR_VALUE);
  use     SENSOR_VALUE_IO;
  --
  task ALARM is
    entry POST_FAULT_IN_SENSOR;
    entry POST_OUT_OF_LIMITS(ON_SENSOR : in SENSOR_NAME);
  end ALARM;
  --
  task RECORDING_DEVICE is
    entry LOG_THE_STATUS(OF_SENSOR   in SENSOR_NAME;
                         WITH_VALUE  in SENSOR_VALUE;
                         WITH_STATE  in SENSOR_STATE);
  end RECORDING_DEVICE;
  --
  task COLLECTION_OF_SENSORS is
    entry DISABLE         (SENSOR       in SENSOR_NAME);
    entry ENABLE          (SENSOR       in SENSOR_NAME);
    entry FORCE_RECORD(OF_SENSOR    in SENSOR_NAME);
    entry SET_THE_LIMITS(FOR_SENSOR in SENSOR_NAME;
                         LOW_LIMIT     in SENSOR_VALUE;
                         HIGH_LIMIT    in SENSOR_VALUE);
  end COLLECTION_OF_SENSORS;
  --
  task TIMER is
    entry INTERRUPT;
    for INTERRUPT use at 16#8E#;
  end TIMER;
  --
  HIGH_BOUND    SENSOR_VALUE;
  LOW_BOUND     SENSOR_VALUE;
  NAME          SENSOR_NAME;
  USER_COMMAND  COMMAND;
  VALUE         SENSOR_VALUE;
  --
  task body ALARM                  is separate;
  task body RECORDING_DEVICE       is separate;
  task body COLLECTION_OF_SENSORS  is separate;
  task body TIMER                  is separate;
```

```
begin
  loop
    begin -- start of a local block with exception handler
      PUT("Enter your command:");
      GET(USER_COMMAND);
      NEW_LINE;
      PUT_LINE("Command accepted");
      case USER_COMMAND is
        when DISABLE =>
                PUT("Enter sensor name:");
                GET(NAME);
                NEW_LINE;
                COLLECTION_OF_SENSORS.DISABLE(SENSOR => NAME);
                PUT_LINE("Sensor disabled");
        when ENABLE =>
                PUT("Enter sensor name:");
                GET(NAME);
                NEW_LINE;
                COLLECTION_OF_SENSORS.ENABLE(SENSOR => NAME);
                PUT_LINE("Sensor enabled");
        when RECORD_STATUS =>
                PUT("Enter sensor name:");
                GET(NAME);
                NEW_LINE;
                COLLECTION_OF_SENSORS.FORCE_RECORD
                        (OF_SENSOR => NAME);
                PUT_LINE("Sensor status set");
        when SET_LIMITS =>
                PUT("Enter sensor name:");
                GET(NAME);
                NEW_LINE;
                PUT ("Enter lower limit:");
                GET (LOW_BOUND);
                NEW_LINE;
                PUT_LINE("Lower limit accepted");
                PUT("Enter upper limit:");
                GET(HIGH_BOUND);
                NEW_LINE;
                PUT_LINE("Upper limit accepted");
                COLLECTION_OF_SENSORS.SET_THE_LIMITS
                        (FOR_SENSOR => NAME,
                         LOW_LIMIT  => LOW_BOUND,
                         HIGH_LIMIT => HIGH_BOUND);
                PUT_LINE("Limits set");
       end case;


       exception
         when DATA_ERROR =>
           PUT_LINE("Illegal entry. . .try again");
       end;
     end loop;
   end MONITOR_TEMPERATURES;
```

```
separate (MONITOR_TEMPERATURES)
task body TIMER is
  MINUTES          constant := 1;
  type INTERVAL is range 0 .. 15;
  TICKS            INTERVAL := 0;
  --
begin
  loop
    accept INTERRUPT do
      TICKS := TICKS + 1;
      if TICKS = 15 * MINUTES then
        for I in SENSOR_NAME
          loop
            select
              COLLECTION_OF_SENSORS.FORCE_RECORD(OF_SENSOR => I);
            or
              delay 5.0;
              ALARM.POST_FAULT_IN_SENSOR;
            end select;
          end loop;
        TICKS := 0;
      end if;
    end INTERRUPT;
  end loop;
end TIMER;
```

```
with SYSTEM;
separate (MONITOR_TEMPERATURES)
task body ALARM is
--
    BITS     : constant := 1;
    WORDS    : constant := 16 * BITS;
    --
    type LIGHT          is (OFF, ON);
    for  LIGHT'SIZE     use 1 * WORDS;
    for  LIGHT          use (OFF => 16#0000#, ON => 16#FFFF#);
    FAULT_LIGHT         : LIGHT := OFF;
    for  FAULT_LIGHT use at 16#0010#;
--
type LIMIT_CHECK            is array (SENSOR_NAME) of LIGHT;
for  LIMIT_CHECK'SIZE       use (SENSOR_NAME'POS(SENSOR_NAME'LAST)
                                + 1) * WORDS;
OUT_OF_LIMITS_LIGHT         : LIMIT_CHECK := LIMIT_CHECK'( others => OFF);
for  OUT_OF_LIMITS_LIGHT use at 16#0011#;
--
begin
  loop
    select
      accept POST_FAULT_IN_SENSOR do
        FAULT_LIGHT := ON;
      end POST_FAULT_IN_SENSOR;
    or
      accept POST_OUT_OF_LIMITS(ON_SENSOR : in SENSOR_NAME) do
        OUT_OF_LIMITS_LIGHT(ON_SENSOR) := ON;
      end POST_OUT_OF_LIMITS;
    end select;
  end loop;
end ALARM;
```

```
with DEVICE_IO;
separate (MONITOR_TEMPERATURES)
task body RECORDING_DEVICE is
  begin
    loop
      accept LOG_THE_STATUS(OF_SENSOR  : in SENSOR_NAME;
                            WITH_VALUE : in SENSOR_VALUE;
                            WITH_STATE : in SENSOR_STATE) do
          DEVICE_IO.PUT(OF_SENSOR);
          DEVICE_IO.PUT(WITH_VALUE);
          DEVICE_IO.PUT(WITH_STATE);
      end LOG_THE_STATUS;
    end loop;
  end RECORDING_DEVICE;
```

```
with SET_PACKAGE, SYSTEM;
separate (MONITOR_TEMPERATURES)
task body COLLECTION_OF_SENSORS is
  --
  BITS    : constant := 1;
  WORDS : constant := 16 * BITS;
  --
  type SENSOR_RECORD is record
    HIGH_LIMIT : SENSOR_VALUE := SENSOR_VALUE'LAST;
    LOW_LIMIT : SENSOR_VALUE := SENSOR_VALUE'FIRST;
    VALUE      : SENSOR_VALUE := SENSOR_VALUE'FIRST;
                        end record;
  type SENSOR_GROUP  is array (SENSOR_NAME) of SENSOR_RECORD;
  SENSOR                : SENSOR_GROUP;
  --
  package SENSOR_SET  is new SET_PACKAGE(UNIVERSE => SENSOR_NAME);
  use SENSOR_SET;
  ACTIVE_SENSORS       : SET := NULL_SET;
  --
  type SENSOR_PORT     is range 0 .. (2 ** WORDS - 1);
  for SENSOR_PORT'SIZE use 1 * WORDS;
  type SENSOR_LIST     is array (SENSOR_NAME) of SENSOR_PORT;
  for SENSOR_LIST'SIZE  use (SENSOR_NAME'POS(SENSOR_NAME'LAST) - 1)
                                                        * WORDS;
  SENSOR_MAP            : SENSOR_LIST;
  for SENSOR_MAP        use at 16#0100#;
  --
begin
  loop
    select
      accept DISABLE(SENSOR : in SENSOR_NAME) do
        ACTIVE_SENSORS := ACTIVE_SENSORS - SENSOR;
      end DISABLE;
    or
      accept ENABLE(SENSOR : in SENSOR_NAME) do
        ACTIVE_SENSORS := ACTIVE_SENSORS + SENSOR;
      end ENABLE;
```

```
    or
      accept FORCE_RECORD(OF_SENSOR  in SENSOR_NAME) do
       if IS_A_MEMBER(OF_SENSOR, OF_SET => ACTIVE_SENSORS) then
         RECORDING_DEVICE.LOG_THE_STATUS(OF_SENSOR
                                         SENSOR(OF_SENSOR).VALUE
                                         WITH_STATE => ENABLED)


       else
         RECORDING_DEVICE.LOG_THE_STATUS(OF_SENSOR,
                                         WITH_VALUE => SENSOR_VALUE'FIRST,
                                         WITH_STATE => DISABLED);
       end if;
      end FORCE_RECORD;
      or
      accept SET_THE_LIMITS(FOR_SENSOR  in SENSOR_NAME;
                            LOW_LIMIT   in SENSOR_VALUE;
                            HIGH_LIMIT  in SENSOR_VALUE) do
       SENSOR(FOR_SENSOR).LOW_LIMIT  := LOW_LIMIT;
       SENSOR(FOR_SENSOR).HIGH_LIMIT := HIGH_LIMIT;
      end SET_THE_LIMITS;
        else
      for I in SENSOR_NAME
        loop
         if IS_A_MEMBER(I, OF_SET => ACTIVE_SENSORS) then
          SENSOR(I).VALUE :=
            (SENSOR_MAP(I) * SENSOR_VALUE(0.5));
          if (SENSOR(I).VALUE < SENSOR(I).LOW_LIMIT) or
              (SENSOR(I).VALUE > SENSOR(I).HIGH_LIMIT) then
            ALARM.POST_OUT_OF_LIMITS(I);
          end if;
         end if ;
        end loop ;
      end select ;
    end loop;
end COLLECTION_OF_SENSORS;
```

REFERENCES

[Arm84] Armitage, J. W. and J. V. Chelini, "Ada Software on Distributed Targets: A Survey of Approaches," GTE Government Systems, Strategic Systems Division TN 84 807.6, October 1984.

[Boo83] Booch, G., ***Software Engineering with Ada***, Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, 1983.

[Che86] Chelini, J. V., E. B. Hudson and S. M. Reidy, "A Preliminary Study of Ada Expansion Ratios," ***ACM Software Engineering Notes***, Vol. 11, No. 1, January 1986, pp. 35-46.

[Chu80] Chu, W. W. et al., "Task Allocation in Distributed Data Processing," ***IEEE Computer***, Vol. 13, No. 11 (1980), pp. 57-69.

[Gar79] Garey, M. R. and David S. Johnson, ***Computers and Intractability: A Guide to the Theory of NP-Completeness***, W. H. Freeman and Company, 1979.

[GTE85] "Task Allocation in Distributed ***Hard*** Real-Time Systems," ***GTE Laboratories Technical Proposal No. RL 5029*** (1985).

[Hoa78] Hoare, C. A. R., "Communicating Sequence Processes," ***Communications of ACM***, Vol. 21, No. 8, August 1978.

[Ker69] Kernighan B. W. and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," ***The Bell System Technical Journal***, February 1970, pp. 291-307.

[Lin81] Lint, B. and T. Agerwala, "Communication Issues in the Design and Analysis of Parallel Algorithms," ***IEEE Transactions on Software Engineering***, Vol. SE-7, No. 2, pp. 174-188.

[Mok84] Mok, A., "The Decomposition of Real-Time System Requirements into Process Models," ***Proceedings of the 1984 Real-Time Systems Symposium***, pp. 125-134.

[Mok85] Mok, A., and S. Sutanthavibul, "Modelling and Scheduling of Dataflow Real-Time Systems," ***Proceedings of the 1985 Real-Time Systems Symposium***, pp. 178-187.

[Pri84] Price, C. C., and S. Krishnaprasad, "Software Allocation Models for Distributed Computing Systems," ***Proceedings of the 1984 Distributed Computing Systems Conference***, pp. 40-18.

[Wea81] Weatherly, R. M., "A Message-based Kernel to Support Ada Tasking," Gensoft Corporation, Pittsburgh, PA, 1981.

END

11-87

DTIC